BLOXY: Providing Transparent and Generic BFT-Based Ordering Services for Blockchains

Signe Rüsch
TU Braunschweig
ruesch@ibr.cs.tu-bs.de

Kai Bleeke
TU Braunschweig
bleeke@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

***Abstract*—With the wide-spread use of blockchain technology, Byzantine fault-tolerant (BFT) protocols are explored as a means to achieve consensus on which transactions should be processed next. BFT protocols are not a one-size-fits-all solution: they should be chosen according to the blockchain's use case, which can range from supply chain management to decentralised storage, requiring specialisation e. g. regarding throughput, latency, or level of decentralisation. Previously, consensus protocols were usually hardcoded into the blockchain infrastructure and could not be exchanged, therefore inhibiting flexible use of an otherwise generic blockchain infrastructure. Hyperledger Fabric claims to provide modular consensus and support for crash-fault and Byzantine fault tolerant protocols. However, integrating a BFT protocol has shown that Fabric's architecture is currently not well-suited for this fault model as it requires substantial changes and thereby breaks Fabric's modularity. This also has to be repeated for each integrated BFT protocol.**

In this paper, we present BLOXY, a blockchain-aware trusted proxy running on the replica that encapsulates all BFT client functionality. BLOXY enables transparent access to generic BFT frameworks and preserves Fabric's modularity even for the Byzantine fault model. It runs inside a trusted execution environment based on Intel's Software Guard Extensions. BLOXY offers blockchain-specific communication mechanisms as well as short-term block storage to handle crashes or disconnects to ensure that all nodes receive block updates. We implemented two BLOXY-based ordering services based on PBFT and the hybrid BFT protocol Hybster. Our evaluation shows that our approach increases the throughput of the ordering component by up to 71 % compared to directly integrated BFT protocols.

***Index Terms*—Byzantine fault tolerance, Blockchain, Hyperledger Fabric**

I. INTRODUCTION

Distributed ledger technology (DLT) is based on the concept of a *blockchain*: the blocks, linked together by including the hash of the predecessor, contain ordered transactions, thereby securing the blockchain against manipulation. Two common use cases for blockchain technologies are cryptocurrencies, such as Bitcoin [1] working with a permissionless blockchain, and supply chain management (SCM) [2], [3], [4], the management of material, information, and services for product manufacturing. In a *permissionless* blockchain, the number of users is usually not known in advance and there is no regulation of nodes entering or leaving the network. For SCM, however, this regulation is necessary as it allows business partners to document transaction flows. Here, *permissioned* blockchains should be employed, i. e. blockchains where access is restricted and the participants are known and identifiable. Different use cases have different requirements e. g. regarding throughput or scalability, leading to use case dependent demands for the consensus protocol that determines how transactions are included in the next block. Allowing blockchain operators to choose the consensus protocol most fitting for their use case would be highly beneficial for the adoption of blockchain platforms in industry as there is no "one-size-fits-all" solution for BFT consensus [5]. Given a certain blockchain platform, however, the consensus protocols so far cannot be adapted according to the individual requirements, i. e. the consensus mechanism is usually hardcoded and cannot easily be exchanged [6], [7], [8], [9], [10].

Platforms that allow operators the choice between consensus protocols during setup are more versatile with regard to their use cases. The permissioned blockchain platform *Hyperledger Fabric* [11] is the only platform where the consensus module, called *ordering service*, is claimed to be "pluggable". This means that different protocols can be chosen or integrated without large effort by implementing a simple interface. Next to a centralised *Solo* ordering service for testing purposes and one based on the crash-fault tolerant (CFT) protocol *Apache Kafka* [12], recent work presented an ordering service supporting the Byzantine fault model based on BFT-SMART [13]. However, there is a fundamental challenge when transitioning from crash-fault tolerance to the Byzantine fault model in Fabric: BFT protocols require certain client functionality, such as majority voting on the results, which would have to be performed on all nodes. As this requires changes to these nodes since parts of the ordering service are now performed elsewhere, the modularity and "pluggability" of the ordering service is therefore impossible to maintain. Additionally, this is necessary for every integrated BFT protocol, i. e. repetitive implementation of identical functionality has to be integrated into Fabric. All BFT replicas have to send their reply, i. e. the created block, to all nodes in Fabric's network instead of just the requesting client as typical in BFT, thus immensely increasing network load. This shows that the integration of BFT protocols into Fabric is not as trivial as the claim of pluggability suggests, and for these reasons, the BFT-SMART implementation had to make certain compromises to support a Byzantine fault model in Fabric, e. g. running the BFT client

Acknowledgements: The authors thank the anonymous reviewers for their valuable feedback. This research was supported by the German Research Council (DFG) under grant no. KA 3171/1-2.

https://doi.org/10.24355/dbbs.084-202001061458-0

on the peer nodes. We aim for a completely modular – and therefore truly pluggable – way to provide CFT and BFT consensus in Fabric via a generic consensus proxy.

In our paper, we present BLOXY: a component to allow modular deployment of BFT-based ordering services in Fabric. It overcomes the challenges that BFT protocols such as BFT-SMART face: (i) We offer full modularity by transparently encapsulating all client functionality inside the BLOXY. (ii) We rely on a trusted subsystem that can only fail by crashing, thereby relaxing the Byzantine fault model to a hybrid one [14], [15], [16], [17], [18], [19], [20], thus allowing us to have trusted client functionality on BFT replicas. BLOXY is running inside a trusted execution environment (TEE) based on Intel's Software Guard Extensions (SGX). (iii) Instead of adapting the communication of BFT frameworks, the BLOXY handles efficient and fault-tolerant block dissemination. Additionally, to ensure the completeness of the blockchain and to offer Byzantine fault tolerance of Fabric's orderers (see § II-B), we present a short-term block storage for efficient retrieval of missing blocks.

We integrated two BFT-based ordering services for Fabric, one based on PBFT [21] and one on the hybrid BFT protocol Hybster [14], using the BLOXY for support of the Byzantine fault model, easy integration, and maintained modularity. With these ordering services, all of Fabric's components can behave arbitrarily within the limit of the Byzantine fault model. Our evaluation shows that the lower communication complexity of BLOXY increases the throughput of the ordering by up to 71 % compared to natively integrated BFT protocols.

Our paper is structured as follows: §II outlines the problem statement, the BLOXY and its implementation are presented in §III and §IV, evaluation results are discussed in §V, §VI gives an overview over related work, and §VII concludes.

II. Background and Problem Statement

Next, we will give an overview of today's BFT protocols for blockchains (§ II-A) as well as an introduction to Fabric and the supported consensus protocols (§ II-B).

A. BFT-Ordering Services for Blockchains

Blockchains are increasingly employed in a growing number of use cases, such as improving transparency in SCM [2], notary or registration services [22], and resilient distributed storage services [23]. However, these use cases have different requirements, leading to use case dependent demands for the consensus protocol, e. g. regarding performance or scalability requirements, trust relations between participants, or whether a trusted third party or trusted administration can be assumed. If malicious behaviour can be excluded and nodes can only fail by crashing, then a crash-fault tolerant protocol for the ordering service suffices. Otherwise the need for Byzantine fault tolerance arises. A consortium of a small number of organisations using a permissioned blockchain for their SCM might have one consensus node for each organisation running in the cloud, possibly in the same data centre, requiring a high-throughput consensus protocol. In a larger supply chain where

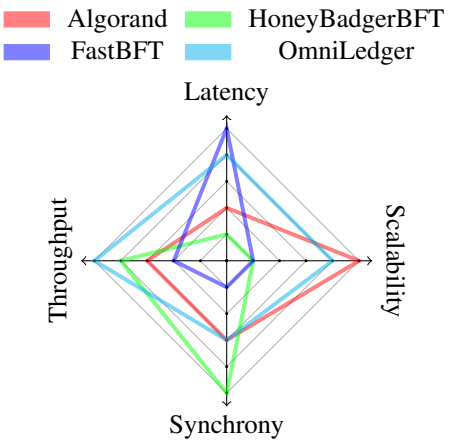


Fig. 1: BFT protocols are optimized for different metrics, ensuring that there is no "one-size-fits-all" solution. Higher is better; higher synchrony equals asynchrony.

the partners are positioned all over the world, the consensus protocol should be able to deal with higher latency and enable geo-replication. Several BFT protocols for blockchains have been presented, targeting these challenges. Fig. 1 shows how different BFT protocols developed for blockchain deployment are optimised for different requirements. Some protocols such as FastBFT [24] are optimised for low-latency communication, whereas others such as OmniLedger [25] aim for high throughput communication. Algorand [26] targets blockchain networks with a large number of participants and is therefore highly scalable. Regarding network conditions, BFT protocols typically assume partial synchrony. HoneyBadgerBFT [27] diverts from that and is also working on asynchronous networks. A more comprehensive study of BFT protocols optimised for blockchains can be found in [28]. Fabric is currently the only blockchain platform providing the versatility of multiple ordering services, each targeting different use cases. Providing operators the freedom to choose the most fitting protocol would greatly improve blockchain adoption in industry settings. However, Fabric's architecture for pluggable consensus is not sufficient for the requirements of a Byzantine fault model. The question how BFT protocols can be integrated into Fabric *generically* so that the advantages of specialised BFT protocols can be exploited according to the specific requirements, is still open. To better understand this problem, we now present the architecture of Fabric with focus on the ordering service in more depth.

B. Hyperledger Fabric

Hyperledger Fabric [11] is a permissioned blockchain framework. Compared to others, the main features of Fabric are a modular architecture that allows e. g. using different consensus protocols or identity management services, and a new *execute-order-validate* paradigm [29]. Traditionally, blockchains follow the order-execute paradigm: transactions are broadcast to all nodes and at some point included in a block, and executed as soon as the block is received. Fabric differs significantly from this model, as transactions are

https://doi.org/10.24355/dbbs.084-202001061458-0

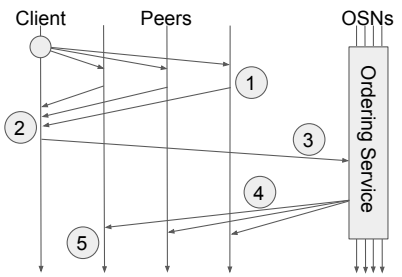


Fig. 2: Transaction flow in Hyperledger Fabric [11]

executed first by a subset of nodes (peers) before being totally ordered by another set of nodes (ordering service). After being ordered, transactions are then broadcast to all nodes and the state change that was computed in the execution phase is validated and applied. This paradigm enables non-deterministic smart contracts, improves performance, and allows Fabric to be independent from cryptocurrencies [11]. State in Fabric is a key-value store which can be modified by invoking Fabric's smart contracts, which are called *chaincodes*, and a Fabric network can contain multiple independent blockchains called channels. There are three groups of nodes in Fabric. *Clients* submit transactions for execution and broadcast them for ordering. *Peers* maintain the blockchain, i. e. they validate incoming transactions and apply the state changes. Both peers and clients are considered potentially Byzantine in Fabric. A subset of the peers is responsible for executing the chaincode transactions; these are called *endorsing peers*. The *ordering service* consists of Ordering Service Nodes (OSNs), also called *orderers*, and is responsible for establishing a total order of the requests. Orderers either run the consensus protocol directly or forward transactions to a cluster of dedicated consensus nodes. The ordering service generally does not maintain the blockchain. To reduce the network load of orderers, peers can exchange blocks via gossip communication.

1) Transaction Flow in Hyperledger Fabric: Fabric's transaction flow, shown in Fig. 2, consists of the three phases execution, ordering, and validation. Chaincodes are invoked by sending a signed transaction proposal to the endorsing peers, the number of which is specified by the chaincode's endorsement policy. Endorsers then simulate the execution of the transaction against their local state (1), i. e. no persistent state changes are performed. After this, the endorser returns an endorsement to the client. The client then collects and appends the endorsements (2) and transmits the transaction to the ordering service, thereby starting the ordering phase (3). The ordering service establishes a total order on all transactions, which are then batched into signed blocks and delivered to the peers (4). The ordering service explicitly does *not* maintain the state of the blockchain or validate transactions. When the ordering is completed, the validation phase is entered. After receiving a block from the ordering service, it is the peer's responsibility to validate the contained transactions (5). The peer checks it against the policy of the invoked chaincode, as well as the validity of the changed data. The block is appended to the peer's copy of the blockchain and all valid state changes are applied to the local state.

2) Consensus in Hyperledger Fabric: Consensus in Hyperledger Fabric is a modular component and can support different levels of fault tolerance, i. e. crash-fault or Byzantine fault tolerance, depending on the chosen protocol. The ordering service itself assumes that all peer and client nodes are potentially Byzantine [11] and implements an atomic broadcast that accepts transactions and ultimately delivers them in blocks to all peers. Independent of the deployed protocol, the ordering service provides the following guarantees [11]:

1. *Agreement*: Two blocks with the same sequence number delivered to different peers should be identical. Forks are not supported.
2. *Hashchain integrity*: Delivered blocks hold the correct hash pointing to the previous block, thus forming a hashchain. Blocks cannot be manipulated without invalidating all hashes in subsequent blocks.
3. *No skipping*: No peer can skip a block. If a peer receives a block, it has already received all previous blocks.
4. *No creation*: Every transaction in a block was created by a client before the block creation.
5. *Liveness*: If a client sends a transaction to the ordering service, then every peer will eventually receive a block containing this transaction.

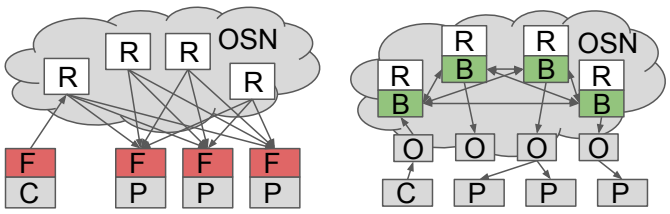
Fabric currently ships with a centralised, non-replicated Solo ordering service for testing purposes providing no fault tolerance, and a distributed, CFT ordering service based on Apache Kafka. Solo requires less computational resources, but presents a single point of failure, whereas Kafka cannot handle Byzantine faults. A BFT ordering service based on BFT-SMaRt has recently been included [13].

C. Integrating BFT-SMaRt into Fabric

BFT protocols require additional functionality on the client side, e. g. the majority voting. Performing this voting on the orderers directly is problematic: if an orderer crashes, then peers can connect to another orderer; however, if it is Byzantine and disseminates corrupted blocks, this might not be detected by the peers, leading to an unsupported fork in the blockchain. To avoid this, the BFT-SMaRt ordering service, whose architecture is shown in Fig. 3a, requires a BFT client library, called *frontend*, running on all peer and client nodes. This frontend also contains Fabric's orderer node. The consensus cluster runs the BFT-SMaRt protocol, and created blocks are signed and disseminated to all frontends. The majority voting can then be performed by every peer individually, provided that it receives $2f + 1$[1] responses from the orderers, and the block can be appended to the blockchain. However, this implementation requires changes to all nodes in the network. In this architecture, Fabric's ordering service is not modular as changing the ordering service requires changes

[1]The authors state that this number can be decreased to $f + 1$ if the peers verify the block's signature.

https://doi.org/10.24355/dbbs.084-202001061458-0



(a) BFT-SMaRt (b) Bloxy

Fig. 3: Architecture of BFT ordering services (P: Peer, C: Client, R: Replica, F: Frontend, B: Bloxy, O: Orderer. Grey components are part of Fabric.)

to peers and clients as well. As peers in Fabric can be Byzantine and the frontend is part of the peer, it might drop blocks or return invalid ones. The *no skipping* property is therefore not fulfilled in the BFT-SMaRt ordering service. Additionally, the implementation is specific to BFT-SMaRt, meaning that while the functionality is required for common BFT protocols, it would have to be repeated for all protocols that Fabric's users choose to integrate. The integration of BFT protocols into Fabric is therefore more complex than suggested by the "pluggability" that was claimed in the original paper [11].

III. The Bloxy

In this paper, we show an approach to provide truly pluggable consensus for different fault models in Fabric, without any changes to its general architecture. For this, we relocate the BFT client functionality to the BFT replica side, thereby containing it inside the ordering service, by leveraging trusted execution to increase the compatibility between an architecture that works well for crash-fault tolerant protocols, but still falls short for Byzantine faults. To provide generic, modular consensus and support for both fault models, we present the Bloxy: a *blockchain-aware trusted proxy*. We define the following requirements:

R1: Support for the Byzantine fault model. Bloxy should allow Fabric to support the Byzantine fault model without compromising the modularity, i. e. placing client functionality of the ordering service onto peers. We aim to keep the ordering service self-contained while still supporting BFT protocols.

R2: Generality and easy integration of BFT protocols. As the BFT client functionality is almost identical across BFT protocols, it would be redundant for each BFT framework to repeat the integration work, especially if doing so compromises Fabric's modularity. Bloxy should provide generality, i. e. it should enable support for multiple BFT protocols. It should therefore be comparably easy to integrate new BFT protocols into Bloxy, see § III-E.

R3: Fulfil Fabric's requirements for ordering services. To be considered a valid and complete system for ordering services, Bloxy-backed solutions should fulfil all requirements of Fabric, see § II-B.

R4: Low performance overhead. The usage of Bloxy should incur only acceptable performance overhead compared to ordering services running without Bloxy, see §V.

A. Bloxy *in a Nutshell*

Fig. 3b shows our approach. The Bloxy is inserted between Fabric's orderer and the consensus cluster, i. e. the BFT replicas, and each replica is equipped with a Bloxy. This means that outside of the consensus cluster, Fabric is now not required to know about the specific consensus protocol, allowing us to completely abstract from it. Compared to the current pluggable consensus of Hyperledger Fabric, which does not work out of the box with BFT protocols without modifications to peers, we go one step further. Using the Bloxy to integrate a new BFT ordering service in Fabric, we do not need to change the underlying Fabric architecture for the Byzantine fault model. We have a *two-layer abstraction*: the Bloxy abstracts from the BFT protocol and handles all communication with the consensus cluster, whereas the orderer abstracts from the underlying blockchain network.

The defined properties for the Fabric ordering service have been described in § II-B. However, the integration of the BFT-SMaRt ordering service shows that Fabric's architecture does not allow the property of *no skipping* to be satisfied for BFT protocols. An orderer or frontend can return a wrong block or not respond at all, which might lead to forks or delays if peers have to query other orderers for blocks, respectively. A wrong block might not be detected by peers.

The Bloxy provides a generic interface for the consensus cluster and behaves identically regardless of the underlying protocol. We are therefore not bound to any specific protocol implementation. Bloxy's extensibility is discussed in § III-E.

The Bloxy completely confines the consensus protocol to the ordering nodes, from receiving and forwarding the request to voting on correct responses and disseminating the new block in the network. As the Bloxy is running inside a TEE, we assume the operations to be trusted, leading to a hybrid fault model: the Bloxy behaves correctly and can only fail by crashing, whereas all other components of the network are untrusted and can behave arbitrarily. The Bloxy can crash or get disconnected from any connected nodes. This is equivalent to a failing BFT replica. Fail-over mechanisms such as DNS round-robin or a load balancer can be used to deal with this and to connect to another Bloxy instance. As the Bloxy handles majority voting in the TEE, we do not require peers to take any further action to verify the created blocks, i. e. to perform the voting themselves as with BFT-SMaRt. This way we maintain Fabric's modularity. Permissioned blockchains require administrators, e. g. from different organisations, for setup, whose roles can be managed via the Membership Service Provider (MSP) and who can mutually monitor each other. We add to this setup process the provisioning of TEEs with cryptographic keys.

To summarise, the tasks of the Bloxy are as follows:

1. Establishing secure connections with orderers and replicas by keeping the TLS session key inside the TEE.
2. Decrypting received transactions from orderers and forwarding them to replicas as BFT requests: as this is done

https://doi.org/10.24355/dbbs.084-202001061458-0

inside the trusted subsystem, modification of transactions or other manipulation is not possible.

3. Collecting, verifying, and comparing the received results, i. e. the blocks, from the replicas in the trusted voting stage, and afterwards disseminating the created block in the network.

B. System Model and Trusted Subsystem

In the Fabric network, all peers and clients are potentially Byzantine, and the ordering service should also support the Byzantine fault model. BFT-SMART e. g. needs to extend the trust of the frontends, placing them in the peer trust domain. The BLOXY takes over all responsibility from the BFT clients and runs it in a TEE on the replica. The TEE allows us to place trust in the performed operations, so that we do not require re-execution on the peers or similar mechanisms. The TEE's integrity can be remotely attested. We therefore assume a hybrid fault model [14], [15], [16], [17], [18], [19], [20]: the ordering service can be Byzantine, whereas all BLOXIES in the system are assumed to behave as expected or fail only by crashing. Nodes can therefore trust the correctness of the block if they receive it from a BLOXY over a secure channel. If a block is received that does not match the winning hash, this means that some fault occurred during the block creation process, leading to a faulty state of the replica. This faulty state will eventually be corrected by the replica when its present state is compared to the state of the other replicas during the regular checkpointing process.

The number of required replicas depends on the protocol requirements. Traditional BFT protocol [21], [30], [31], [32] require up to $3f+1$ replicas to tolerate up to f faults. Protocols that utilise trusted components [14], [15], [16], [17], [18], [19], [20] and thus assume a hybrid fault model, can reduce this number to $2f + 1$ as they can use the TEE to prevent equivocation. One challenge of trusted execution is to keep the trusted code base (TCB) as small as possible to reduce the probability of failures [17]. For this, we place only limited parts of the BLOXY inside the TEE. The secure connection establishment is performed by generating and storing the TLS key as well as performing cryptographic operations inside the TEE so that the key never leaves it. The actual connection handling can be done in the untrusted part of the application. After receiving a request, it is decrypted inside the TEE and the validity is checked, before it is immediately authenticated using HMACs to prevent any modification once it is passed outside of the TEE again. When the BLOXY has collected a sufficient number of replies, it performs the majority voting again inside the TEE, where it assembles and encrypts the correct reply to all connected orderers. The actual transmission can be performed by the application's untrusted part as the TLS keys are not available there, so manipulation is prevented.

C. Transaction Flow with BLOXY

The message flow of a transaction using the BLOXY is shown in Fig. 4. As mentioned previously, every replica is equipped with a BLOXY instance running on the same machine

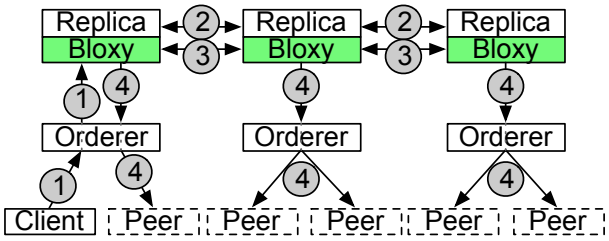


Fig. 4: Transaction flow of a BLOXY-based BFT ordering service

inside a TEE, which possesses a key pair and whose authenticity can be remotely attested. A client sends its transaction to an orderer who forwards it to the BLOXY (1). Neither node has to be aware of the underlying consensus protocol. The BLOXY now assumes the role of the BFT client, forms a BFT request containing the transaction, and forwards it to the leader of the BFT protocol. It then sends an ACK to the blockchain client that its transaction has been received and that the client does not need to retransmit this transaction. If this step fails, the BLOXY sends a NAK. The consensus nodes now deterministically establish a total order on the received transactions according to the protocol (2). After either a certain limit of requests or the total block size has been reached, the transactions are batched, and the block is created and signed for authenticity. This is done by invoking the block cutting mechanism, which runs as a replicated service. To increase fault tolerance and availability, we let every BLOXY vote on the replies: the BLOXY broadcasts the hash of the received block to all other BLOXIES, which have to receive the full signed block from their connected replica (3). The BLOXIES verify the signatures of the replicas and perform the trusted voting, i. e. check that at least $f + 1$ identical blocks have been received. This voting is necessary to ensure that the block has been created correctly, to fulfil the *agreement* property of permissioned blockchains and to prevent forks. We require only $f+1$ identical responses as the BLOXY is running inside a TEE and we can therefore prevent equivocation. Once the BLOXIES have voted on the block, it is disseminated in the network by sending it to all connected orderers which then deliver it to the peers (4).

Fabric has two kinds of configuration transactions which are placed in dedicated blocks: (i) new channel transactions which when received in a block by the orderer will cause them to create a new channel, i. e. blockchain, and (ii) channel configuration updates. Configuration transactions are ordered the same way as regular transactions. While this works well for new channel transactions, manual oversight is required for configuration updates to ensure that they are applied correctly at every orderer. This can be an additional task of the blockchain administrators designated via the MSP.

To summarise, the orderer forwards the transactions to the BLOXIES, which then handle all BFT-specific operations and return the created blocks after successful majority voting. This encapsulation of client functionality allows for completely abstracting the consensus protocol from the blockchain archi-

https://doi.org/10.24355/dbbs.084-202001061458-0

tecture, and the BFT protocol is not aware of any blockchain-specific behaviour as well.

D. BLOXY Communication

The communication of BFT protocols used in blockchains differs from that of traditional BFT protocols. The reply, i. e. the created block, has to be disseminated to all nodes in the network instead of sending a reply to one single client. Several protocols use gossip protocols for this [33], [26]. For the BFT-SMART ordering service, the framework was extended to send the block to every connected frontend. With BLOXY, we have a *two-phase broadcasting mechanism* for blocks. First, all BLOXIES need to receive the created block: as described in § III-C, the BLOXY that forwarded the client transactions receives the created block from the replica and then sends it to the other BLOXIES. The BLOXIES can also exchange only hashes of the received blocks to reduce messaging overhead with potentially large blocks. With this optimization, however, every BLOXY needs to receive the created block from the connected replica, as it is necessary that each BLOXY possesses at least one full block to perform the trusted voting. Having each replica send the reply to the connected BLOXY might require minor modification of the framework, as this is not standard behaviour for BFT protocols.

In the second stage after the voting, the block is then broadcast by each BLOXY to all connected orderers to disseminate the block in the network. As the BLOXY only sends the block after the majority voting has been performed instead of multiple replies from the replicas to all peers, we drastically decrease the messaging overhead. This can be seen in Fig. 3, and is later shown in our evaluation in § V-C. The orderers can connect to the BLOXIES using round-robin or other load balancing mechanisms to evenly distribute the load across all available BLOXIES. This dissemination means that the *no skipping* property of Fabric ordering services can be fulfilled, as the block is not only available on one node, but every node can query the orderers.

BLOXY Short-Term Block Storage. With the exception of *no skipping*, the properties of Fabric's ordering services can be fulfilled by any BFT protocol. Fabric depends on the orderers to disseminate the block in the rest of the network. The orderers store a certain range of previous blocks for the peers to query; however, up to f orderers can behave Byzantine. This means that the orderers can suffer from disconnects, crashes, or arbitrary behaviour, and we cannot guarantee that only correct orderers store and receive the created blocks. The correctness of blocks is guaranteed and can be verified by the BLOXY's trusted voting and the replicas' signatures, any manipulation of the block would thus be detected. The orderers can only misbehave by not delivering the block to peers, i. e. by behaving as if crashed. To satisfy the *no skipping* property, peers can query orderers for created blocks. If a peer only contacts one orderer, it might not detect e. g. a crash and that it is missing blocks. To ensure that *no skipping* is fulfilled and Byzantine orderers in Fabric can be tolerated, the replicas store a certain number of past blocks in a *Short-Term Block Storage*. A benign orderer can then query this short-term storage for any missing blocks, and blocks are only deleted from the short-term storage, when at least $f+1$ orderers have acknowledged its reception. This satisfies our requirement *R3* as all properties of Fabric's ordering service are fulfilled.

Fabric now has to handle crash faults on the orderer side, especially in the Byzantine fault model, which is currently only partially addressed by simply querying other orderers for blocks. To handle reliable reception of blocks as well as correct voting, the BFT-SMART ordering service merged the orderer and peer nodes in the frontend to assume the same fault model, i. e. that orderers can be assumed to run correctly. Our short-term block storage allows reliable dissemination of blocks in the network. A block is stored until a certain threshold of orderers has acknowledged it, thereby offering Byzantine fault tolerance for orderers.

Blocks that have passed the majority voting inside the TEE are stored inside the short-term storage, which can be queried by orderers and peers in case they are missing a recent block, e. g. because they re-connect after a longer crash. The blocks, once they are created, are not changed by further write requests. The short-term storage is running as a replicated service on the BFT replicas and can be queried via BFT read requests. Read requests for stored blocks are forwarded to the replicas, and the BLOXY receives either the block if it is stored, or a NAK otherwise. After voting on the result, it is passed to the requesting node. As the BFT cluster should not store the blockchain and to limit the required storage space, blocks are removed from the short-term storage after they have been received by a sufficient number of nodes in the network, i. e. by at least $f+1$ nodes to offer crash-fault tolerance. Orderers are therefore required to acknowledge the reception of a block. If orderers or peers request a block that has been purged from short-term storage, they in addition to the NAK receive a list of a fixed number of node addresses that have previously acknowledged the block to query them directly and receive the missing block.

The short-term storage is not inside the TEE as the memory of current TEE technologies such as Intel SGX is highly limited, which is problematic as the default block size in Fabric is ≈2MB. Our evaluation results in § V-C show that the Byzantine fault tolerance comes at the cost of higher latency.

E. Providing Generic Consensus

Using the BLOXY and trusted execution, we can unobtrusively bridge the conceptual gap between a crash fault model and a Byzantine fault model without introducing complex changes to Fabric's architecture. The orderers are now acting purely as proxies, forwarding transactions to the BLOXIES in the BFT cluster and storing the blockchain to be queried by peers. Assuming that f from $3f+1$ orderers are Byzantine, we can still reliably and transparently provide the BFT client functionality in the BLOXY and peers can connect to another orderer if they detect misbehaviour.

With regard to employed BFT protocols, we are not limited to any specific protocol, though some minor changes to the

https://doi.org/10.24355/dbbs.084-202001061458-0

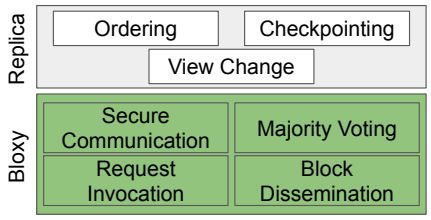


Fig. 5: Functionality of the BFT replica and of the BFT client as performed by the BLOXY

BFT frameworks might be necessary for integration. The BFT framework can include the BLOXY as a library. An overview of the functionalities performed by the BFT replica and the BLOXY is shown in Fig. 5. The BLOXY is needed for establishing secure TLS connections with orderers and BFT replicas, achieved by placing the en- and decryption routines of the communication inside the TEE. It also transforms the blockchain transactions into BFT client requests, for which it has to know the message format of the protocol. Lastly, the BLOXY has to perform the majority voting inside the TEE, after which the valid reply is encrypted and sent to all connected orderers via the TLS connections. The essential parts of the BFT protocol, however, remain unmodified, namely the ordering, view change, and checkpointing subprotocols. As BFT protocols generally provide similar interfaces and pose similar requirements to the applications, the underlying BFT protocol can therefore be changed at will. The BLOXY only forwards the transformed requests to the replicas and requires a certain amount of replies for the majority voting, otherwise it is completely independent from the BFT protocol. Our approach is orthogonal to the recent improvements of BFT protocols for blockchains, which offers the opportunity to take advantage of improved BFT protocols that are the most useful for Fabric and the given use case. We therefore satisfy requirement *R2* regarding generality and easy integration.

IV. IMPLEMENTATION

We now give insight into the implementation of the BLOXY with details on Intel SGX (§ IV-A), as well as the BLOXY-based ordering services based on PBFT and Hybster (§ IV-B).

A. BLOXY *Implementation*

The trusted execution environment used to run the BLOXY is based on Intel SGX, an extension to Intel's x86 processor instruction set. SGX enables the generation of trusted compartments, called *enclaves*. Enclaves are used to ensure integrity and confidentiality of security-sensitive computations, while the remainder of the machine is assumed to be untrusted or malicious. Confidentiality of code and data is ensured using memory encryption, while checksums are used for integrity. Enclaves can only be entered or exited using pre-defined entry points. We use the SGX SDK version 2.1 for the definition of calls inside the trusted enclave (*ecalls*) and calls from inside to the untrusted outside (*ocalls*). We utilize a modified version of the TaLoS [34] library for trusted cryptographic operations inside the enclave, which offers the same interface as LibreSSL for cryptographic operations that are then performed securely inside the enclave. The modification includes running TaLoS completely inside the enclave as it is only accessed via the trusted BLOXY. This library is used to handle bidirectional TLS connections securely inside the enclave.

Intel provides a remote attestation service that can verify the validity of enclaves. After attestation, the enclave can be provisioned with cryptographic keys, e. g. a private key for establishing the TLS connections.

BLOXY is implemented in C/C++. To keep the TCB small, we aim to keep the number of lines of code (LoC) within the enclave to a minimum. BLOXY's trusted part inside the enclave consists of 251 kLoC: 219 kLoC for TaLoS, 12 kLoC for BLOXY logic including 2 kLoC for a protobuf implementation [35] ported to the enclave, and 20 kLoC for the SGX SDK. The code base of TaLoS is an upper bound estimation: it offers the same interface as LibreSSL, but BLOXY only uses a limited subset of its functionality. Additionally, we have kept the interface as minimal as possible, containing only 20 *ecalls* and no *ocalls*. The *ecalls* include sanity checks on input values to prevent Iago [36] attacks or time-of-check-to-time-of-use attacks [37], as well as bound checking of pointers to ensure that they point to enclave memory.

B. BLOXY*-based Ordering Services*

Hybster [14] is a hybrid BFT protocol, i. e. it uses a trusted subsystem for message authentication to prevent equivocation. It therefore requires only $2f + 1$ replicas to tolerate f Byzantine faults. Hybster is implemented in Java and, with the consensus-oriented parallelisation scheme [38], is optimised to fully exploit multi-core CPUs. This BFT protocol therefore achieves a high throughput, which makes it suitable for deployment in blockchain use cases. The trusted subsystem is also based on Intel SGX and therefore well-suited to be combined with the BLOXY. The interface between the BLOXY written in C/C++ and Hybster written in Java is implemented using the Java Native Interface (JNI). PBFT [21] was implemented in previous Fabric versions, but not in the current one. Using BLOXY, it can be re-integrated with minimal effort.

The replicated service running on the replicas is the *block cutting*. When the service receives the ordered requests, it checks whether the maximum number of transactions for one block or the maximum block size has been reached. If this is the case, the block cutter bundles these transactions into one block which is then disseminated. The service maintains all necessary Fabric channels, i. e. blockchains, adding new channels or modifying configurations if configuration transactions are received. In this case, all previous transactions are flushed into a new block, before the configuration transaction is placed in a dedicated block. For Hybster, the signing of a block as well as the block dissemination are performed in separate *pillars*, i. e. threads.

We integrated the BLOXY into Hyperledger Fabric v1.2, released in July 2018, utilising the Fabric Java SDK. The communication between Fabric nodes, e. g. the orderers and peers, is done via gRPC, whereas the communication between

https://doi.org/10.24355/dbbs.084-202001061458-0

the orderers and the BLOXY as well as the BLOXY and the Hybster or PBFT replicas is done via TLS. For the short-term block storage, it is necessary to know to which channel a block belongs. A minimal protobuf implementation [35] was ported inside the enclave to enable extraction of the channel name.

V. EVALUATION

We evaluate the security and performance of the BLOXY-based ordering services against Fabric's Solo, Kafka, and the natively integrated BFT-SMART protocols. Our results show that BLOXY-based ordering services (i) are resilient against multiple attacks, (ii) achieve higher throughput and lower or similar latency as BFT-SMART, (iii) do not affect the performance of a full Fabric setup, and (iv) provide higher fault tolerance of blocks due to the short-term block storage. We therefore completely satisfy our requirement *R4* regarding performance overhead.

A. Experimental Setup

We use a cluster of ten machines divided into three classes for our evaluation. Class (A) consists of four machines equipped with SGX-capable 4-core Xeon v5 CPUs and 32GB of memory; they are running the consensus cluster. Class (B) are five machines equipped with 4-core Xeon v2 CPUs and 16GB of memory. Four are running a peer and orderer node each, while the other runs the clients. One class (C) machine is used for orchestration; it has a 4-core Intel Xeon E3 CPU and 16GB of memory. All machines have hyper-threading enabled and are connected via a 10Gpbs network. Class (A) runs Ubuntu 18.04, (B) and (C) run Ubuntu 14.04.

B. Security Evaluation

Inspired by [39], we present attacks on BLOXY and our countermeasures.

1) Side-channel attacks: As side-channel attacks are not considered in the SGX attack model, they are out of scope. However, previously suggested mitigation techniques for SGX can be applied to BLOXY [40], [41], [42].

2) Bypassing BLOXY: To prevent malicious replicas from bypassing the BLOXY and sending corrupted blocks directly to the orderers, we use TLS connections between orderers and BLOXIES. The session keys are stored securely inside the TEE.

3) Interface attacks: The interface of an enclave can be attacked in an attempt to gain access to BLOXY's secrets, e. g. with Iago attacks [36]. We have hardened the interface with checks on input and return values.

4) Denial-of-service (DoS) and flooding: Faulty replicas could perform DoS attacks by not running the BLOXY, or flood correct replicas and orderers with corrupted blocks. BLOXY can use existing techniques [43] for prevention, e. g. by limiting the number of requests a client can issue simultaneously.

5) Block signatures: Blocks are signed by the replicas, i. e. orderers are incapable of manipulating them without immediate detection. They would need access to the replicas' private keys and certificates. However, even then they could only discard or reorder transactions in a block, not manipulate specific transactions as they are again signed by the client. The trust in the blocks therefore comes from the block signature and the security of the replicas' private keys.

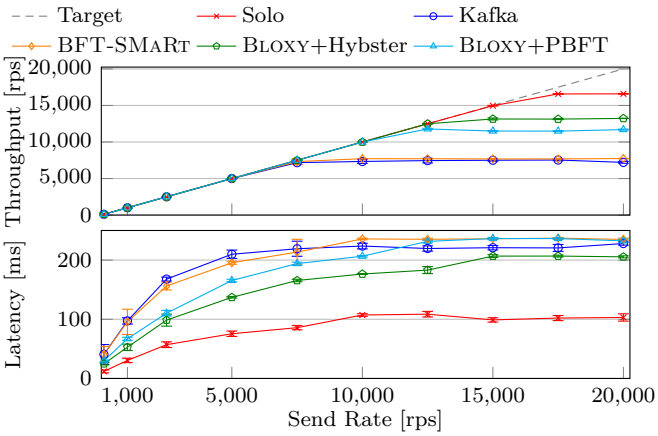


Fig. 6: Throughput and latency of the ordering service (5 repetitions, 10 s each)

6) Malicious orderers: The only faults orderers can exhibit are delivering blocks with invalid signatures or no block at all, which are both equivalent to a crashed orderer. The same applies for withholding submitted transactions, which a client can re-issue. Peers can detect this behaviour and as a failover mechanism reconnect with another orderer. This requires that at least one correct orderer has the created blocks, which can be ensured with the short-term block storage.

C. Microbenchmarks

Ordering Service. We have measured the latency and throughput of the ordering services for an increasing send rate for messages of 1KB. The client sends transactions and receives the created blocks, which contain 100 transactions. The latency is the average latency of all transactions in the block from sending until delivery of the block. The results, which can be seen in Fig. 6, show that BLOXY-based Hybster and PBFT have 71 % and 51 % higher throughput compared to BFT-SMART, and only 20 % and 29.5 % less throughput than the centralised Solo, respectively. The performance of Kafka is limited as block generation and signing are not parallelised, which was necessary in previous versions of Fabric. For send rates higher than 15,000 rps, the latency of both BLOXY-backed PBFT and BFT-SMART is 125.7 % higher than Solo's; in comparison, the latency of BLOXY-backed Hybster is only 99.5 % higher than Solo. In this measurement, we can see that the decreased communication overhead of BLOXY, which we achieve by performing the majority voting on the replicas inside the consensus cluster, is advantageous for throughput and latency of the ordering service.

Short-Term Block Storage. We measure the latency when an orderer (i) successfully queries the short-term block storage for a missing block, (ii) queries the short-term storage, does not receive the block, and queries another orderer based on the returned address (see § III-D), or (iii) queries other orderers

https://doi.org/10.24355/dbbs.084-202001061458-0

	BLOXY		(iii) Orderer
	(i) Storage Hit	(ii) Storage Miss	
Duration	21.79 ms	7.19 ms	7.28 ms

TABLE I: Latency for fetching missing blocks from short-term block storage or from other orderer nodes (5 repetitions)

	Solo	Kafka	BFT-SMART	BLOXY Hybster	BLOXY PBFT
s_L	0.9 s	0.96 s	1.09 s	0.81 s	0.85 s
SR_{s_L}	1,400 rps	1,300 rps	1,200 rps	1,500 rps	1,300 rps
s_T	147.64 rps	360.28 rps	155.5 rps	120.58 rps	122.85 rps
SR_{s_T}	1,400 rps	1,600 rps	1,600 rps	1,500 rps	1,500 rps

TABLE II: Maximum standard deviation for latency (s_L) and throughput (s_T) and the corresponding send rate ($SR_{s_{L/T}}$).

randomly. The results are shown in Table I. Even if the block is not stored in the short-term block storage, the BLOXY responds with the address of at least one orderer that has acknowledged the block reception which can then be queried. The latency of a storage miss is therefore still smaller than querying other orderers at random, which might require multiple attempts before the block is received. When the block is stored on the replicas, the BLOXY queries it via a BFT read request, receives three blocks as response, and votes on them before sending it to the requesting orderer. The induced latency is therefore higher than directly querying orderers; however, this is justified by the fault tolerance offered to the orderers with the short-term block storage. The recovery strategy can be configured in the orderer, thus allowing the low-latency querying of orderers to be standard behaviour and querying the short-term block storage as a fail-safe mechanism.

D. Macrobenchmarks

We use the Hyperledger Caliper [44] project to compare the ordering services regarding throughput and latency within a full Fabric network, again with increasing send rates. In the network, we have twelve clients, two peers in the same organisation, and two CAs. The consensus cluster consists of four replicas, except for Hybster which as a hybrid protocol is run with three. We execute Caliper's *Simple* benchmark. The results are shown in Fig. 7. The latency of all ordering services starts to increase rapidly with a send rate higher than 1,000 rps; similarly, the throughput starts to decrease at 1,300 rps. The maximum standard deviations for send rates above 1,200 rps are given in Table II. This behaviour is similar across all ordering services, leading to the conclusion that the ordering service is not the bottleneck of Fabric's network. The results are in line with previous results for evaluating Fabric with Caliper [45].

VI. RELATED WORK

BFT protocols for blockchains. Most BFT protocols target the permissioned setting; however, compared to the Proof-of-Work (PoW) consensus, they are severely restricted regarding scalability. Examples of protocols for permissioned

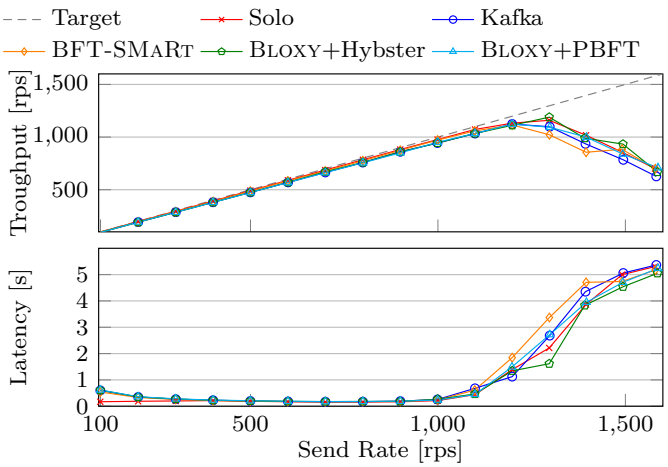


Fig. 7: Throughput and latency of a complete Fabric network with different ordering services (5 repetitions, each 20 s)

blockchains are ByzCoin [46], Stellar Consensus [47], Ripple [9], HoneyBadgerBFT [27], and Gosig [33]. Algorand [26] is a BFT protocol designed for permissionless blockchains. Here, every decision and voting step is done by secretly choosing a comparatively small subset of nodes, and each node is chosen via Proof-of-Stake (PoS). Several blockchain platforms already employ BFT protocols as basis for their consensus. Hyperledger Fabric v0.6 used PBFT, while v1.0 was extended by a BFT-SMART ordering service [13]. Tendermint [10] uses a round-based voting similar to PBFT. Quorum [8] has two consensus protocols, based on Raft and PBFT; and Hyperledger Iroha [7] uses the PBFT-based protocol YAC. Some consensus protocols make use of trusted execution. In the "Proof of Elapsed Time" (PoET) [48] protocol, an SGX-based timer is employed, similar to the "Proof of Luck" (PoL) [49] protocol. As BLOXY is designed for easy integration of BFT protocols, suitable protocols targeting permissioned blockchains can be integrated, allowing to make use of their individual advantages. The approaches of these protocols are therefore orthogonal to ours.

SGX as trusted proxy. There have been several approaches to use TEEs for establishing secure proxies [50], [39], [51]. BFT systems are not wide-spread compared to crash-fault tolerant systems, since they require several changes to the clients, e. g. the additional voting step to reconcile multiple responses. Changing all client implementations especially for commonly used protocols such as HTTP and IMAP would not be feasible. Instead, Troxy [39] presents a trusted proxy running inside a TEE that abstracts the BFT client functionality from the actual client device, instead shifting it to the BFT replicas. We follow a similar approach of shifting BFT client functionality; however, the BLOXY is tailored for the blockchain use case by including blockchain-specific communication, the short-term block storage, as well as integration into Hyperledger Fabric.

Database servers running in the cloud are vulnerable to several exploits that can cause the client to receive wrong results. VeritasDB [50] presents a key-value store which guarantees

https://doi.org/10.24355/dbbs.084-202001061458-0

integrity to the client by offering a network proxy that employs an SGX enclave. This enclave performs integrity checks to reduce the necessary trust in cloud providers. LibSEAL [51] is an audit library to detect integrity violations on user data by creating non-repudiable audit logs inside an enclave. To capture all service messages in the logs and prevent tampering, the TLS connections are terminated inside the enclave.

VII. Conclusion

In this paper, we presented BLOXY, a consensus proxy for Hyperledger Fabric. BLOXY encapsulates the client functionality of BFT protocols, thereby enabling easy integration of consensus protocols in Fabric and allowing operators to choose protocols suitable for their use cases. Furthermore, BLOXY-backed ordering services maintain Fabric's modularity. Our evaluation shows that due to BLOXY's lower communication complexity, we can achieve up to 71 % higher throughput for the ordering compared to directly integrated BFT protocols.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] T. Bocek, B. Rodrigues, T. Strasser, and B. Stiller, "Blockchains Everywhere – A Use-Case of Blockchains in the Pharma Supply-Chain," in *IEEE IM*, 2017.

[3] F. Tian, "An Agri-Food Supply Chain Traceability System for China based on RFID & Blockchain Technology," in *ICSSSM*, 2016.

[4] K. Korpela, J. Hallikas, and T. Dahlberg, "Digital Supply Chain Transformation Toward Blockchain Integration," in *HICSS*, 2017.

[5] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT Protocols Under Fire," in *NSDI*, 2008.

[6] Chain Inc. (2017) Chain Protocol Whitepaper. [Online]. Available: https://chain.com/docs/1.2/protocol/papers/whitepaper

[7] The Linux Foundation. (2018) Hyperledger Iroha. [Online]. Available: https://www.hyperledger.org/projects/iroha

[8] JPMorgan Chase & Co. (2016) Quorum Whitepaper v0.1. [Online]. Available: https://github.com/jpmorganchase/quorum-docs/blob/master/QuorumWhitepaperv0.1.pdf

[9] D. Schwartz, N. Youngs, A. Britto *et al.*, "The Ripple Protocol Consensus Algorithm," *Ripple Labs Inc White Paper*, 2014.

[10] J. Kwon, "Tendermint: Consensus Without Mining," https://tendermint.com/static/docs/tendermint.pdf, 2014.

[11] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart *et al.*, "Hyperledger Fabric: a Distributed Operating System for Permissioned Blockchains," in *EuroSys*, 2018.

[12] K. Christidis. (2016) A Kafka-based Ordering Service for Fabric. [Online]. Available: https://docs.google.com/document/d/19JihmW-8blTzN99lAubOfseLUZqdrB6sBR0HsRgCAnY/edit

[13] J. Sousa, A. Bessani, and M. Vukolic, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform," in *DSN*, 2018.

[14] J. Behl, T. Distler, and R. Kapitza, "Hybrids on Steroids: SGX-based High Performance BFT," in *EuroSys*, 2017.

[15] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested Append-Only Memory: Making Adversaries Stick to Their Word," in *ACM SIGOPS OSR*, 2007.

[16] T. Distler, C. Cachin, and R. Kapitza, "Resource-Efficient Byzantine Fault Tolerance," *IEEE TC*, 2016.

[17] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat *et al.*, "CheapBFT: Resource-Efficient Byzantine Fault Tolerance," in *EuroSys*, 2012.

[18] D. Levin, J. Douceur, J. Lorch, and T. Moscibroda, "TrInc: Small Trusted Hardware for Large Distributed Systems," in *NSDI*, 2009.

[19] B. Vavala, N. Neves, and P. Steenkiste, "Securing Passive Replication Through Verification," in *SRDS*, 2015.

[20] G. S. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine Fault-Tolerance," *IEEE TC*, 2013.

[21] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI*, 1999.

[22] H. A. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan, "An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design," in *WEIS*, 2015.

[23] S. Ruj, M. S. Rahman, A. Basu, and S. Kiyomoto, "BlockStore: A Secure Decentralized Storage Framework on Blockchain," in *AINA*, 2018.

[24] J. Liu, W. Li, G. Karame, and N. Asokan, "Scalable Byzantine Consensus via Hardware-assisted Secret Sharing," *IEEE TC*, 2018.

[25] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," https://eprint.iacr.org/2017/406.pdf, 2017.

[26] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *SOSP*, 2017.

[27] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The Honey Badger of BFT Protocols," in *CCS*, 2016.

[28] C. Berger and H. P. Reiser, "Scaling Byzantine Consensus: A Broad Analysis," in *SERIAL*, 2018.

[29] M. Vukolić, "Rethinking Permissioned Blockchains," in *BCC*, 2017.

[30] M. Castro and B. Liskov, "Proactive Recovery in a Byzantine Fault-Tolerant System," in *OSDI*, 2000.

[31] R. Kotla and M. Dahlin, "High Throughput Byzantine Fault Tolerance," *DSN*, 2004.

[32] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services," *ACM SIGOPS OSR*, 2003.

[33] P. Li, G. Wang, X. Chen, and W. Xu, "Gosig: Scalable Byzantine Consensus on Adversarial Wide Area Network for Blockchains," *arXiv preprint arXiv:1802.01315*, 2018.

[34] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn *et al.*, "TaLoS: Secure and transparent TLS termination inside SGX enclaves," Technical Report 2017/5, Imperial College London, Tech. Rep., 2017.

[35] P. Aimonen. Nanobp – Protocol Buffers With Small Code Size. [Online]. Available: https://jpa.kapsi.fi/nanopb/

[36] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," in *ASPLOS*, 2013.

[37] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *ESORICS*, 2016.

[38] J. Behl, T. Distler, and R. Kapitza, "Consensus-Oriented Parallelization: How to Earn Your First Million," in *Middleware*, 2015.

[39] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, "Troxy: Transparent Access to Byzantine Fault-Tolerant Systems," in *DSN*, 2018.

[40] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *NDSS*, 2017.

[41] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-Grained Control Flow Inside SGX Enclaves With Branch Shadowing," *arXiv preprint arXiv:1611.06952*, 2016.

[42] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *NDSS*, 2017.

[43] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," in *NSDI*, 2009.

[44] The Linux Foundation. (2018) Hyperledger Caliper. [Online]. Available: https://www.hyperledger.org/projects/caliper

[45] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance Characterization of Hyperledger Fabric," in *CVCBT*, 2018.

[46] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin Security and Performance With Strong Consistency via Collective Signing," in *USENIX Security*, 2016.

[47] D. Mazieres, "The Stellar Consensus Protocol: A Federated Model for Internet-Level Consensus," *Stellar Development Foundation*, 2015.

[48] (2017) Proof of Elapsed Time. [Online]. Available: https://sawtooth.hyperledger.org/docs/core/nightly/0-8/introduction.html

[49] M. Milutinovic, W. He, H. Wu, and M. Kanwal, "Proof of Luck: An Efficient Blockchain Consensus Protocol," in *SysTEX*, 2016.

[50] R. Sinha and M. Christodorescu, "VeritasDB: High Throughput Key-Value Store with Integrity," *IACR Cryptology ePrint Archive*, 2018.

[51] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn *et al.*, "LibSEAL: Revealing Service Integrity Violations Using Trusted Execution," in *EuroSys*, 2018.

https://doi.org/10.24355/dbbs.084-202001061458-0